PCI IDE Controller

From OSDev Wiki

The factual accuracy of this article or section is disputed.
Please see the relevant discussion on the talk page.

IDE is a keyword which refers to the electrical specification of the cables which connect ATA drives (like hard drives) to another device. The drives use the ATA (Advanced Technology Attachment) interface. An IDE cable also can terminate at an IDE card connected to PCI.

ATAPI is an extension to ATA (recently renamed to PATA) which adds support for the SCSI command set.

Contents

Parallel/Serial ATA/ATAPI

IDE can connect up to 4 drives. Each drive can be one of the following:

- ATA (Serial): Used for most modern hard drives.
- ATA (Parallel): Commonly used for hard drives.
- ATAPI (Serial): Used for most modern optical drives.
- ATAPI (Parallel): Commonly used for optical drives.

Accessing an ATA/PATA drive works the same way as accessing a SATA drive. This also implicitly states that accessing a PATAPI ODD is the same as accessing a SATAPI ODD. An IDE driver does not need to know whether a drive is parallel or serial, it only has to know whether it's using ATA or ATAPI.

IDE Interface

If you open your case up and take a look at your motherboard, you will most likely see one or two (or possibly more) of the slots that can be seen in the picture to the right.

The white and green ports are IDE ports, also known as *channels*. In this example there are both primary and secondary IDE channels which only PATA can be connected to; this means that it only supports PATA/PATAPI drives.

Each port can have a PATA cable connected to it (see the image on the right). One master drive, or two drives (master and slave), can be connected to one PATA cable. So that leaves us with the following possibilities:

- Primary Master Drive.
- Primary Slave Drive.
- Secondary Master Drive.
- Secondary Slave Drive.

Each drive can be either PATA or PATAPI.

Serial IDE

Almost every modern motherboard has a Serial IDE channel which allows SATA and SATAPI Drives to be connected to it. There are 4 Serial IDE Ports (you can see these in the photo to the right). Each port is connected to a drive with a SATA Cable. Basically (looking at the pictures), you can only have one drive connected to the Serial IDE port. Each pair of ports (every 2 ports) form one channel.

Serial IDE also has a few possibilities:

- Primary Master, also called SATA1.
- Primary Slave, also called SATA2.
- Secondary Master, also called SATA3.
- Secondary Slave, also called SATA4.

Detecting a PCI IDE Controller

Each IDE controller appears as a device on the PCI bus. If the class code is 0x01 (Mass Storage Controller) and the subclass code is 0x1, (IDE) this device is an IDE Device. The IDE device only uses five BARs out of the six

- BAR0: Base address of primary channel (I/O space), if it is 0x0 or 0x1, the port is 0x1F0.
- BAR1: Base address of primary channel control port (I/O space), if it is 0x0 or 0x1, the port is 0x3F6.
- BAR2: Base address of secondary channel (I/O space), if it is 0x0 or 0x1, the port is 0x170.
- BAR3: Base address of secondary channel control port, if it is 0x0 or 0x1, the port is 0x376.
- BAR4: Bus Master IDE; refers to the base of I/O range consisting of 16 ports. Each 8 ports controls DMA on the primary and secondary channel respectively.

A parallel IDE controller will use IRQs 14 and 15; a serial IDE uses only one IRQ. To read this IRQ, we look through the device's PCI configuration space:

```
outl((1 << 31) | (bus << 16) | (device << 11) | (func << 8) | 8, 0xCF8);
if ((inl(0xCFC) >> 16) != 0xFFFF) { // If device exists (class isn't 0xFF
```

```
    // Check if this device needs an IRQ assignment:
    outl((1 << 31) | (bus << 16) | (device << 11) | (func << 8) | 0x3C, 0x
    outb(0xFE, 0xCFC); // Change the IRQ field to 0xFE
    outl((1 << 31) | (bus << 16) | (device << 11) | (func << 8) | 0x3C, 0x
    if ((inl(0xCFC) & 0xFF) == 0xFE) {
       // This device needs an IRQ assignment.
    } else {
       // The device doesn't use IRQs, check if this is an Parallel IDE:
       if (class == 0x01 && subclass == 0x01 && (ProgIF == 0x8A || ProgIF
          // This is a Parallel IDE Controller which uses IRQs 14 and 15.
       }
    }
}
```

Detecting IDE Drives

To initialise the IDE driver, we call ide_initialise:

```
void ide_initialize(unsigned int BAR0, unsigned int BAR1, unsigned int BA
unsigned int BAR4) {
```

If you only want to support the parallel IDE, you can use these parameters:

```
ide_initialize(0x1F0, 0x3F6, 0x170, 0x376, 0x000);
```

We can assume that BAR4 is 0x0 because we are not going to use it yet. We will return to ide_initialize, which searches for drives connected to the IDE. Before we go into this function, we should write some support functions and definitions which will help us a lot.

```
#define ATA_SR_BSY     0x80
#define ATA_SR_DRDY    0x40
#define ATA_SR_DF      0x20
#define ATA_SR_DSC     0x10
#define ATA_SR_DRQ     0x08
#define ATA_SR_CORR    0x04
#define ATA_SR_IDX     0x02
#define ATA_SR_ERR     0x01
```

The Command/Status Port returns a bit mask referring to the status of a channel when read.

```
#define ATA_ER_BBK      0x80
#define ATA_ER_UNC      0x40
#define ATA_ER_MC       0x20
```

```
#define ATA_ER_IDNF     0x10
#define ATA_ER_MCR      0x08
#define ATA_ER_ABRT     0x04
#define ATA_ER_TK0NF    0x02
#define ATA_ER_AMNF     0x01
```

The Features/Error Port, which returns the most recent error upon read, has these possible bit masks

```
#define ATA_CMD_READ_PIO          0x20
#define ATA_CMD_READ_PIO_EXT      0x24
#define ATA_CMD_READ_DMA          0xC8
#define ATA_CMD_READ_DMA_EXT      0x25
#define ATA_CMD_WRITE_PIO         0x30
#define ATA_CMD_WRITE_PIO_EXT     0x34
#define ATA_CMD_WRITE_DMA         0xCA
#define ATA_CMD_WRITE_DMA_EXT     0x35
#define ATA_CMD_CACHE_FLUSH       0xE7
#define ATA_CMD_CACHE_FLUSH_EXT   0xEA
#define ATA_CMD_PACKET            0xA0
#define ATA_CMD_IDENTIFY_PACKET   0xA1
#define ATA_CMD_IDENTIFY          0xEC
```

When you write to the Command/Status port, you are executing one of the commands above.

```
#define      ATAPI_CMD_READ       0xA8
#define      ATAPI_CMD_EJECT      0x1B
```

The commands above are for ATAPI devices, which will be understood soon.

ATA_CMD_IDENTIFY_PACKET and ATA_CMD_IDENTIFY return a buffer of 512 bytes called the identification space; the following definitions are used to read information from the identification space.

```
#define ATA_IDENT_DEVICETYPE   0
#define ATA_IDENT_CYLINDERS    2
#define ATA_IDENT_HEADS        6
#define ATA_IDENT_SECTORS      12
#define ATA_IDENT_SERIAL       20
#define ATA_IDENT_MODEL        54
#define ATA_IDENT_CAPABILITIES 98
#define ATA_IDENT_FIELDVALID   106
#define ATA_IDENT_MAX_LBA      120
#define ATA_IDENT_COMMANDSETS  164
#define ATA_IDENT_MAX_LBA_EXT  200
```

When you select a drive, you should specify the interface type and whether it is the master or slave:

```
#define IDE_ATA        0x00
#define IDE_ATAPI      0x01

#define ATA_MASTER     0x00
#define ATA_SLAVE      0x01
```

Task File is a range of 8 ports which are offsets from BAR0 (primary channel) and/or BAR2 (secondary channel). To exemplify:

- BAR0 + 0 is first port.
- BAR0 + 1 is second port.
- BAR0 + 2 is the third

```
#define ATA_REG_DATA       0x00
#define ATA_REG_ERROR      0x01
#define ATA_REG_FEATURES   0x01
#define ATA_REG_SECCOUNT0  0x02
#define ATA_REG_LBA0       0x03
#define ATA_REG_LBA1       0x04
#define ATA_REG_LBA2       0x05
#define ATA_REG_HDDEVSEL   0x06
#define ATA_REG_COMMAND    0x07
#define ATA_REG_STATUS     0x07
#define ATA_REG_SECCOUNT1  0x08
#define ATA_REG_LBA3       0x09
#define ATA_REG_LBA4       0x0A
#define ATA_REG_LBA5       0x0B
#define ATA_REG_CONTROL    0x0C
#define ATA_REG_ALTSTATUS  0x0C
#define ATA_REG_DEVADDRESS 0x0D
```

The ALTSTATUS/CONTROL port returns the alternate status when read and controls a channel when written to.

- For the primary channel, ALTSTATUS/CONTROL port is BAR1 + 2.
- For the secondary channel, ALTSTATUS/CONTROL port is BAR3 + 2.

We can now say that each channel has 13 registers. For the primary channel, we use these values:

- Data Register: BAR0 + 0; // Read-Write
- Error Register: BAR0 + 1; // Read Only
- Features Register: BAR0 + 1; // Write Only
- SECCOUNT0: BAR0 + 2; // Read-Write
- LBA0: BAR0 + 3; // Read-Write
- LBA1: BAR0 + 4; // Read-Write
- LBA2: BAR0 + 5; // Read-Write
- HDDEVSEL: BAR0 + 6; // Read-Write, used to select a drive in the channel.
- Command Register: BAR0 + 7; // Write Only.
- Status Register: BAR0 + 7; // Read Only.
- Alternate Status Register: BAR1 + 2; // Read Only.

- Control Register: BAR1 + 2; // Write Only.
- DEVADDRESS: BAR1 + 3; // I don't know what is the benefit from this register.

The map above is the same with the secondary channel, but it uses BAR2 and BAR3 instead of BAR0 and BAR1.

```
// Channels:
#define      ATA_PRIMARY      0x00
#define      ATA_SECONDARY    0x01

// Directions:
#define      ATA_READ      0x00
#define      ATA_WRITE     0x01
```

We have defined everything needed by the driver, now lets move to an important part. We said that

- BAR0 is the start of the I/O ports used by the primary channel.
- BAR1 is the start of the I/O ports which control the primary channel.
- BAR2 is the start of the I/O ports used by secondary channel.
- BAR3 is the start of the I/O ports which control secondary channel.
- BAR4 is the start of 8 I/O ports controls the primary channel's Bus Master IDE.
- BAR4 + 8 is the Base of 8 I/O ports controls secondary channel's Bus Master IDE.

So we can make this global structure:

```
struct IDEChannelRegisters {
   unsigned short base;  // I/O Base.
   unsigned short ctrl;  // Control Base
   unsigned short bmide; // Bus Master IDE
   unsigned char  nIEN;  // nIEN (No Interrupt);
} channels[2];
```

We also need a buffer to read the identification space into, we need a variable that indicates if an irq is invoked or not, and finally we need an array of 6 words [12 bytes] for ATAPI Drives:

```
unsigned char ide_buf[2048] = {0};
unsigned static char ide_irq_invoked = 0;
unsigned static char atapi_packet[12] = {0xA8, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

We said the the IDE can contain up to 4 drives:

```
struct ide_device {
   unsigned char  Reserved;    // 0 (Empty) or 1 (This Drive really exist
   unsigned char  Channel;     // 0 (Primary Channel) or 1 (Secondary Cha
   unsigned char  Drive;       // 0 (Master Drive) or 1 (Slave Drive).
   unsigned short Type;        // 0: ATA, 1:ATAPI.
```

```
   unsigned short Signature;   // Drive Signature
   unsigned short Capabilities;// Features.
   unsigned int   CommandSets; // Command Sets Supported.
   unsigned int   Size;        // Size in Sectors.
   unsigned char  Model[41];   // Model in string.
} ide_devices[4];
```

When we read a register in a channel, like STATUS Register, it is easy to execute:

```
ide_read(channel, ATA_REG_STATUS);

unsigned char ide_read(unsigned char channel, unsigned char reg) {
   unsigned char result;
   if (reg > 0x07 && reg < 0x0C)
      ide_write(channel, ATA_REG_CONTROL, 0x80 | channels[channel].nIEN);
   if (reg < 0x08)
      result = inb(channels[channel].base + reg - 0x00);
   else if (reg < 0x0C)
      result = inb(channels[channel].base  + reg - 0x06);
   else if (reg < 0x0E)
      result = inb(channels[channel].ctrl  + reg - 0x0A);
   else if (reg < 0x16)
      result = inb(channels[channel].bmide + reg - 0x0E);
   if (reg > 0x07 && reg < 0x0C)
      ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN);
   return result;
}
```

We also need a function for writing to registers:

```
void ide_write(unsigned char channel, unsigned char reg, unsigned char da
   if (reg > 0x07 && reg < 0x0C)
      ide_write(channel, ATA_REG_CONTROL, 0x80 | channels[channel].nIEN);
   if (reg < 0x08)
      outb(channels[channel].base  + reg - 0x00, data);
   else if (reg < 0x0C)
      outb(channels[channel].base  + reg - 0x06, data);
   else if (reg < 0x0E)
      outb(channels[channel].ctrl  + reg - 0x0A, data);
   else if (reg < 0x16)
      outb(channels[channel].bmide + reg - 0x0E, data);
   if (reg > 0x07 && reg < 0x0C)
      ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN);
}
```

To read the identification space, we should read the Data Register as a double word 128 times. We can then copy them to our buffer.

```
void ide_read_buffer(unsigned char channel, unsigned char reg, unsigned i
                     unsigned int quads) {
   /* WARNING: This code contains a serious bug. The inline assembly tras
    *          ESP for all of the code the compiler generates between th
    *          assembly blocks.
    */
   if (reg > 0x07 && reg < 0x0C)
      ide_write(channel, ATA_REG_CONTROL, 0x80 | channels[channel].nIEN);
   asm("pushw %es; movw %ds, %ax; movw %ax, %es");
   if (reg < 0x08)
      insl(channels[channel].base  + reg - 0x00, buffer, quads);
   else if (reg < 0x0C)
      insl(channels[channel].base  + reg - 0x06, buffer, quads);
   else if (reg < 0x0E)
      insl(channels[channel].ctrl  + reg - 0x0A, buffer, quads);
   else if (reg < 0x16)
      insl(channels[channel].bmide + reg - 0x0E, buffer, quads);
   asm("popw %es;");
   if (reg > 0x07 && reg < 0x0C)
      ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN);
}
```

When we send a command, we should wait for 400 nanosecond, then read the Status port. If the Busy bit is on, we should read the status port again until the Busy bit is 0; then we can read the results of the command. This operation is called "Polling". We can also use IRQs instead of polling.

After many commands, if the Device Fault bit is set, there is a failure; if DRQ is not set, there is an error. If the ERR bit is set, there is an error which is described in Error port.

```
unsigned char ide_polling(unsigned char channel, unsigned int advanced_ch

   // (I) Delay 400 nanosecond for BSY to be set:
   // -------------------------------------------------
   for(int i = 0; i < 4; i++)
      ide_read(channel, ATA_REG_ALTSTATUS); // Reading the Alternate Stat

   // (II) Wait for BSY to be cleared:
   // -------------------------------------------------
   while (ide_read(channel, ATA_REG_STATUS) & ATA_SR_BSY)
      ; // Wait for BSY to be zero.

   if (advanced_check) {
      unsigned char state = ide_read(channel, ATA_REG_STATUS); // Read St

      // (III) Check For Errors:
```

```
      // ------------------------------------------------
      if (state & ATA_SR_ERR)
         return 2; // Error.

      // (IV) Check If Device fault:
      // ------------------------------------------------
      if (state & ATA_SR_DF)
         return 1; // Device Fault.

      // (V) Check DRQ:
      // ------------------------------------------------
      // BSY = 0; DF = 0; ERR = 0 so we should check for DRQ now.
      if ((state & ATA_SR_DRQ) == 0)
         return 3; // DRQ should be set

   }

   return 0; // No Error.

}
```

If there is an error, we have a function which prints errors on screen:

```
unsigned char ide_print_error(unsigned int drive, unsigned char err) {
   if (err == 0)
      return err;

   printk("IDE:");
   if (err == 1) {printk("- Device Fault\n     "); err = 19;}
   else if (err == 2) {
      unsigned char st = ide_read(ide_devices[drive].channel, ATA_REG_ERR
      if (st & ATA_ER_AMNF)   {printk("- No Address Mark Found\n     ");
      if (st & ATA_ER_TK0NF)   {printk("- No Media or Media Error\n     "
      if (st & ATA_ER_ABRT)   {printk("- Command Aborted\n     ");      e
      if (st & ATA_ER_MCR)   {printk("- No Media or Media Error\n     ");
      if (st & ATA_ER_IDNF)   {printk("- ID mark not Found\n     ");
      if (st & ATA_ER_MC)   {printk("- No Media or Media Error\n     ");
      if (st & ATA_ER_UNC)   {printk("- Uncorrectable Data Error\n     ")
      if (st & ATA_ER_BBK)   {printk("- Bad Sectors\n     ");       err =
   } else  if (err == 3)           {printk("- Reads Nothing\n     "); err
     else  if (err == 4)  {printk("- Write Protected\n     "); err = 8;}
   printk("- [%s %s] %s\n",
      (const char *[]){"Primary", "Secondary"}[ide_devices[drive].channel
      (const char *[]){"Master", "Slave"}[ide_devices[drive].drive], // S
      ide_devices[drive].model);

   return err;
}
```

Now let's return to the initialization function:

```
void ide_initialize(unsigned int BAR0, unsigned int BAR1, unsigned int BA
unsigned int BAR4) {

   int j, k, count = 0;

   // 1- Detect I/O Ports which interface IDE Controller:
   channels[ATA_PRIMARY  ].base  = (BAR0 & 0xFFFFFFFC) + 0x1F0 * (!BAR0);
   channels[ATA_PRIMARY  ].ctrl  = (BAR1 & 0xFFFFFFFC) + 0x3F6 * (!BAR1);
   channels[ATA_SECONDARY].base  = (BAR2 & 0xFFFFFFFC) + 0x170 * (!BAR2);
   channels[ATA_SECONDARY].ctrl  = (BAR3 & 0xFFFFFFFC) + 0x376 * (!BAR3);
   channels[ATA_PRIMARY  ].bmide = (BAR4 & 0xFFFFFFFC) + 0; // Bus Master
   channels[ATA_SECONDARY].bmide = (BAR4 & 0xFFFFFFFC) + 8; // Bus Master
```

Then we should disable IRQs in both channels by setting bit 1 (nIEN) in the Control port:

```
   // 2- Disable IRQs:
   ide_write(ATA_PRIMARY  , ATA_REG_CONTROL, 2);
   ide_write(ATA_SECONDARY, ATA_REG_CONTROL, 2);
```

Now we need to check for drives which could be connected to each channel. We will select the master drive of each channel, and send the ATA_IDENTIFY command (which is supported by ATA Drives). If there's no error, there are values returned in registers which determine the type of Drive; if no drive is present, there will be strange values.

Notice that if bit 4 in HDDEVSEL is set to 1, we are selecting the slave drive, if set to 0, we are selecting the master drive.

```
   // 3- Detect ATA-ATAPI Devices:
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++) {

         unsigned char err = 0, type = IDE_ATA, status;
         ide_devices[count].Reserved = 0; // Assuming that no drive here.

         // (I) Select Drive:
         ide_write(i, ATA_REG_HDDEVSEL, 0xA0 | (j << 4)); // Select Drive
         sleep(1); // Wait 1ms for drive select to work.

         // (II) Send ATA Identify Command:
         ide_write(i, ATA_REG_COMMAND, ATA_CMD_IDENTIFY);
         sleep(1); // This function should be implemented in your OS. whi
                   // it is based on System Timer Device Driver.
```

```
         // (III) Polling:
         if (ide_read(i, ATA_REG_STATUS) == 0) continue; // If Status = 0

         while(1) {
            status = ide_read(i, ATA_REG_STATUS);
            if ((status & ATA_SR_ERR)) {err = 1; break;} // If Err, Devic
            if (!(status & ATA_SR_BSY) && (status & ATA_SR_DRQ)) break; /
         }

         // (IV) Probe for ATAPI Devices:

         if (err != 0) {
            unsigned char cl = ide_read(i, ATA_REG_LBA1);
            unsigned char ch = ide_read(i, ATA_REG_LBA2);

            if (cl == 0x14 && ch ==0xEB)
               type = IDE_ATAPI;
            else if (cl == 0x69 && ch == 0x96)
               type = IDE_ATAPI;
            else
               continue; // Unknown Type (may not be a device).

            ide_write(i, ATA_REG_COMMAND, ATA_CMD_IDENTIFY_PACKET);
            sleep(1);
         }

         // (V) Read Identification Space of the Device:
         ide_read_buffer(i, ATA_REG_DATA, (unsigned int) ide_buf, 128);

         // (VI) Read Device Parameters:
         ide_devices[count].Reserved     = 1;
         ide_devices[count].Type         = type;
         ide_devices[count].Channel      = i;
         ide_devices[count].Drive        = j;
         ide_devices[count].Signature    = *((unsigned short *)(ide_buf +
         ide_devices[count].Capabilities = *((unsigned short *)(ide_buf +
         ide_devices[count].CommandSets  = *((unsigned int *)(ide_buf + A

         // (VII) Get Size:
         if (ide_devices[count].CommandSets & (1 << 26))
            // Device uses 48-Bit Addressing:
            ide_devices[count].Size   = *((unsigned int *)(ide_buf + ATA_
         else
            // Device uses CHS or 28-bit Addressing:
            ide_devices[count].Size   = *((unsigned int *)(ide_buf + ATA_

         // (VIII) String indicates model of device (like Western Digital
         for(k = 0; k < 40; k += 2) {
            ide_devices[count].Model[k] = ide_buf[ATA_IDENT_MODEL + k + 1
            ide_devices[count].Model[k + 1] = ide_buf[ATA_IDENT_MODEL + k
         ide_devices[count].Model[40] = 0; // Terminate String.
```

```
            count++;
         }

      // 4- Print Summary:
      for (i = 0; i < 4; i++)
         if (ide_devices[i].Reserved == 1) {
            printk(" Found %s Drive %dGB - %s\n",
               (const char *[]){"ATA", "ATAPI"}[ide_devices[i].Type],
               ide_devices[i].Size / 1024 / 1024 / 2,               /* Size
               ide_devices[i].Model);
         }
   }
```

Read/Write From ATA Drive

Now we're moving to a slightly more advanced part, it is to read and write from/to an ATA drive. There is 3 ways of addressing a sector:

- CHS (Cylinder-Head-Sector): an old way of addressing sectors in ATA drives, all ATA drives should support this way of addressing.
- LBA28: Accessing a sector by its 28-bit LBA address. All ATA drives should support this way of addressing, the problem with LBA28 Addressing is that it only allows access 128GB to be accessed, so if the disk is bigger than 128GB, it should support the LBA48 Feature Set.
- LBA48: Accessing a sector by its 48-bit LBA address. As we use integers in GCC, our maximum address in this tutorial is 32-bit long, which allows accessing a drive with a size of up to 2TB.

So we can conclude an algorithm to determine which type of Addressing we are going to use:

```
if (No LBA support)
   Use CHS.
else if (the LBA Sector Address > 0x0FFFFFFF)
   Use LBA48.
else
   Use LBA28.
```

Reading the buffer may be done by polling or DMA. PIO: After sending the command to read or write sectors, we read or write to the Data Port (as words). This is the same way of reading identification space. DMA: After sending the command, you should wait for an IRQ, while you are waiting, Buffer is written directly to memory automatically.

We are going to use PIO as it is less complex.

We can conclude also this table:

```
      /* ATA/ATAPI Read/Write Modes:
       * ++++++++++++++++++++++++++++++++
       *  Addressing Modes:
       *  ================
       *    - LBA28 Mode.     (+)
```

```
    *   - LBA48 Mode.     (+)
    *   - CHS.            (+)
    *  Reading Modes:
    *  ================
    *   - PIO Modes (0 : 6)       (+) // Slower than DMA, but not a probl
    *   - Single Word DMA Modes (0, 1, 2).
    *   - Double Word DMA Modes (0, 1, 2).
    *   - Ultra DMA Modes (0 : 6).
    *  Polling Modes:
    *  ================
    *   - IRQs
    *   - Polling Status   (+) // Suitable for Singletasking
    */
```

There is something needed to be expressed here, I have told before that Task-File is like that:

- Register 0: [Word] Data Register. (Read-Write).
- Register 1: [Byte] Error Register. (Read).
- Register 1: [Byte] Features Register. (Write).
- Register 2: [Byte] SECCOUNT0 Register. (Read-Write).
- Register 3: [Byte] LBA0 Register. (Read-Write).
- Register 4: [Byte] LBA1 Register. (Read-Write).
- Register 5: [Byte] LBA2 Register. (Read-Write).
- Register 6: [Byte] HDDEVSEL Register. (Read-Write).
- Register 7: [Byte] Command Register. (Write).
- Register 7: [Byte] Status Register. (Read).

So each register between 2 to 5 should be 8-bits long. Really each of them are 16-bits long.

- Register 2: [Bits 0-7] SECCOUNT0, [Bits 8-15] SECOUNT1
- Register 3: [Bits 0-7] LBA0, [Bits 8-15] LBA3
- Register 4: [Bits 0-7] LBA1, [Bits 8-15] LBA4
- Register 5: [Bits 0-7] LBA2, [Bits 8-15] LBA5

The word [(SECCOUNT1 << 8) | SECCOUNT0] expresses the number of sectors which can be read when you access by LBA48. When you access in CHS or LBA28, SECCOUNT0 only expresses number of sectors.

- LBA0 makes up bits 0 : 7 of the LBA address when you read in LBA28 or LBA48; it can also be the sector number of CHS.
- LBA1 makes up bits 8 : 15 of the LBA address when you read in LBA28 or LBA48; it can also be the low byte of the cylinder number of CHS.
- LBA2 makes up bits 16 : 23 of the LBA address when you read in LBA28 or LBA48; it can also be the high byte of the cylinder number of CHS.
- LBA3 makes up bits 24 : 31 of the LBA48 address.
- LBA4 makes up bits 32 : 39 of the LBA48 address.
- LBA5 makes up bits 40 : 47 of LBA48 address.

Notice that the LBA0, 1 and 2 registers are 24 bits long in total, which is not enough for LBA28; the higher 4-bits can be written to the lower 4-bits of the HDDEVSEL register.

Also notice that if bit 6 of this register is set, we are going to use LBA, if not, we are going to use CHS. There is a mode which is called extended CHS.

Lets go into the code:

```
unsigned char ide_ata_access(unsigned char direction, unsigned char drive
                             unsigned char numsects, unsigned short selec
```

This function reads/writes sectors from ATA-Drive. If direction is 0 we are reading, else we are writing.

- drive is the drive number which can be from 0 to 3.
- lba is the LBA address which allows us to access disks up to 2TB.
- numsects is the number of sectors to be read, it is a char, as reading more than 256 sector immediately may performance issues. If numsects is 0, the ATA controller will know that we want 256 sectors.
- selector is the segment selector to read from, or write to.
- edi is the offset in that segment.

```
   unsigned char lba_mode /* 0: CHS, 1:LBA28, 2: LBA48 */, dma /* 0: No L
   unsigned char lba_io[6];
   unsigned int  channel      = ide_devices[drive].Channel; // Read the C
   unsigned int  slavebit      = ide_devices[drive].Drive; // Read the Dr
   unsigned int  bus = channels[channel].Base; // Bus Base, like 0x1F0 wh
   unsigned int  words      = 256; // Almost every ATA drive has a sector
   unsigned short cyl, i;
   unsigned char head, sect, err;
```

We don't need IRQs, so we should disable it to prevent problems from happening. We said before that if bit 1 of the Control Register (which is called nIEN bit), is set, no IRQs will be invoked from any drives on this channel, either master or slave.

```
ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN = (ide_irq_inv
```

Now lets read the parameters:

```
   // (I) Select one from LBA28, LBA48 or CHS;
   if (lba >= 0x10000000) { // Sure Drive should support LBA in this case
                            // giving a wrong LBA.
      // LBA48:
      lba_mode  = 2;
      lba_io[0] = (lba & 0x000000FF) >> 0;
      lba_io[1] = (lba & 0x0000FF00) >> 8;
      lba_io[2] = (lba & 0x00FF0000) >> 16;
      lba_io[3] = (lba & 0xFF000000) >> 24;
```

```
      lba_io[4] = 0; // LBA28 is integer, so 32-bits are enough to access
      lba_io[5] = 0; // LBA28 is integer, so 32-bits are enough to access
      head      = 0; // Lower 4-bits of HDDEVSEL are not used here.
   } else if (ide_devices[drive].Capabilities & 0x200)  { // Drive suppor
      // LBA28:
      lba_mode  = 1;
      lba_io[0] = (lba & 0x00000FF) >> 0;
      lba_io[1] = (lba & 0x000FF00) >> 8;
      lba_io[2] = (lba & 0x0FF0000) >> 16;
      lba_io[3] = 0; // These Registers are not used here.
      lba_io[4] = 0; // These Registers are not used here.
      lba_io[5] = 0; // These Registers are not used here.
      head      = (lba & 0xF000000) >> 24;
   } else {
      // CHS:
      lba_mode  = 0;
      sect      = (lba % 63) + 1;
      cyl       = (lba + 1  - sect) / (16 * 63);
      lba_io[0] = sect;
      lba_io[1] = (cyl >> 0) & 0xFF;
      lba_io[2] = (cyl >> 8) & 0xFF;
      lba_io[3] = 0;
      lba_io[4] = 0;
      lba_io[5] = 0;
      head      = (lba + 1  - sect) % (16 * 63) / (63); // Head number is
   }
```

Now we are going to choose the way of reading the buffer [PIO or DMA]:

```
   // (II) See if drive supports DMA or not;
   dma = 0; // We don't support DMA
```

Lets poll the Status port while the channel is busy:

```
   // (III) Wait if the drive is busy;
   while (ide_read(channel, ATA_REG_STATUS) & ATA_SR_BSY)
      ; // Wait if busy.
```

The HDDDEVSEL register now looks like this:

- Bits 0 : 3: Head Number for CHS.
- Bit 4: Slave Bit. (0: Selecting Master Drive, 1: Selecting Slave Drive).
- Bit 5: Obsolete and isn't used, but should be set.
- Bit 6: LBA (0: CHS, 1: LBA).
- Bit 7: Obsolete and isn't used, but should be set.

Lets write all these information to the register, while the obsolete bits are set (0xA0):

```
   // (IV) Select Drive from the controller;
   if (lba_mode == 0)
      ide_write(channel, ATA_REG_HDDEVSEL, 0xA0 | (slavebit << 4) | head)
   else
      ide_write(channel, ATA_REG_HDDEVSEL, 0xE0 | (slavebit << 4) | head)
```

Let's write the parameters to registers:

```
   // (V) Write Parameters;
   if (lba_mode == 2) {
      ide_write(channel, ATA_REG_SECCOUNT1,   0);
      ide_write(channel, ATA_REG_LBA3,   lba_io[3]);
      ide_write(channel, ATA_REG_LBA4,   lba_io[4]);
      ide_write(channel, ATA_REG_LBA5,   lba_io[5]);
   }
   ide_write(channel, ATA_REG_SECCOUNT0,   numsects);
   ide_write(channel, ATA_REG_LBA0,   lba_io[0]);
   ide_write(channel, ATA_REG_LBA1,   lba_io[1]);
   ide_write(channel, ATA_REG_LBA2,   lba_io[2]);
```

If you are using LBA48 and want to write to the LBA0 and LBA3 registers, you should write LBA3 to Register 3, then write LBA0 to Register 3. ide_write function makes it quite simple, refer to the function and you will fully understand the code.

Now, we have a great set of commands described in ATA/ATAPI-8 Specification, we should choose the suitable command to execute:

```
   // (VI) Select the command and send it;
   // Routine that is followed:
   // If ( DMA & LBA48)   DO_DMA_EXT;
   // If ( DMA & LBA28)   DO_DMA_LBA;
   // If ( DMA & LBA28)   DO_DMA_CHS;
   // If (!DMA & LBA48)   DO_PIO_EXT;
   // If (!DMA & LBA28)   DO_PIO_LBA;
   // If (!DMA & !LBA#)   DO_PIO_CHS;
```

There isn't a command for doing CHS with DMA.

```
   if (lba_mode == 0 && dma == 0 && direction == 0) cmd = ATA_CMD_READ_PI
   if (lba_mode == 1 && dma == 0 && direction == 0) cmd = ATA_CMD_READ_PI
   if (lba_mode == 2 && dma == 0 && direction == 0) cmd = ATA_CMD_READ_PI
   if (lba_mode == 0 && dma == 1 && direction == 0) cmd = ATA_CMD_READ_DM
   if (lba_mode == 1 && dma == 1 && direction == 0) cmd = ATA_CMD_READ_DM
   if (lba_mode == 2 && dma == 1 && direction == 0) cmd = ATA_CMD_READ_DM
   if (lba_mode == 0 && dma == 0 && direction == 1) cmd = ATA_CMD_WRITE_F
```

```
   if (lba_mode == 1 && dma == 0 && direction == 1) cmd = ATA_CMD_WRITE_F
   if (lba_mode == 2 && dma == 0 && direction == 1) cmd = ATA_CMD_WRITE_F
   if (lba_mode == 0 && dma == 1 && direction == 1) cmd = ATA_CMD_WRITE_D
   if (lba_mode == 1 && dma == 1 && direction == 1) cmd = ATA_CMD_WRITE_D
   if (lba_mode == 2 && dma == 1 && direction == 1) cmd = ATA_CMD_WRITE_D
   ide_write(channel, ATA_REG_COMMAND, cmd);               // Send the Co
```

This ATA_CMD_READ_PIO command is used for reading in LBA28 or CHS, and the IDE controller refers to bit 6 of the HDDEVSEL register to find out the mode of reading (LBA or CHS).

After sending the command, we should poll, then we read/write a sector, then we should poll, then we read/write a sector, until we read/write all sectors needed, if an error has happened, the function will return a specific error code.

Notice that after writing, we should execute the CACHE FLUSH command, and we should poll after it, but without checking for errors.

```
   if (dma)
      if (direction == 0);
         // DMA Read.
      else;
         // DMA Write.
   else
      if (direction == 0)
         // PIO Read.
      for (i = 0; i < numsects; i++) {
         if (err = ide_polling(channel, 1))
            return err; // Polling, set error and exit if there is.
         asm("pushw %es");
         asm("mov %%ax, %%es" : : "a"(selector));
         asm("rep insw" : : "c"(words), "d"(bus), "D"(edi)); // Receive D
         asm("popw %es");
         edi += (words*2);
      } else {
      // PIO Write.
         for (i = 0; i < numsects; i++) {
            ide_polling(channel, 0); // Polling.
            asm("pushw %ds");
            asm("mov %%ax, %%ds"::"a"(selector));
            asm("rep outsw"::"c"(words), "d"(bus), "S"(edi)); // Send Dat
            asm("popw %ds");
            edi += (words*2);
         }
         ide_write(channel, ATA_REG_COMMAND, (char []) {   ATA_CMD_CACHE_
                        ATA_CMD_CACHE_FLUSH,
                        ATA_CMD_CACHE_FLUSH_EXT}[lba_mode]);
         ide_polling(channel, 0); // Polling.
      }
```

```
   return 0; // Easy, isn't it?
}
```

Reading from an ATAPI Drive

Let's move to an easier part - reading from an ATAPI drive. I will not make the function write to an ATAPI drive, because the write Operation is very complex and is outside of the scope of this tutorial.

An ATAPI drive is different from an ATA drive, as it uses the SCSI command set, not the ATA command set. Parameters are sent as packets, so it is called the ATA-Packet Interface [ATAPI].

Notice also that ATAPI drives always use IRQs, you can't disable them. We should create a function which waits for an IRQ to be caused:

```
void ide_wait_irq() {
   while (!ide_irq_invoked)
      ;
   ide_irq_invoked = 0;
}
```

When an IRQ happens, the following function should be executed by ISR:

```
void ide_irq() {
   ide_irq_invoked = 1;
}
```

ide_wait_irq will go into a while loop, which waits for the variable ide_irq_invoked to be set, then clears it.

```
unsigned char ide_atapi_read(unsigned char drive, unsigned int lba, unsig
         unsigned short selector, unsigned int edi) {
```

- drive is the drive number, which is from 0 to 3.
- lba is the LBA address.
- numsects is the number of sectors. It should always be 1, and if you want to read more than one sector, re-execute this function with th updated LBA address.
- selector is the Segment Selector.
- edi is the offset in the selector.

Let's read the parameters of the drive:

```
   unsigned int   channel  = ide_devices[drive].Channel;
   unsigned int   slavebit = ide_devices[drive].Drive;
   unsigned int   bus      = channels[channel].Base;
```

```
    unsigned int   words    = 1024; // Sector Size. ATAPI drives have a se
    unsigned char  err;
    int i;
```

We need IRQs:

```
    // Enable IRQs:
    ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN = ide_irq_i
```

Let's setup the SCSI Packet, which is 6 words (12 bytes) long:

```
    // (I): Setup SCSI Packet:
    // ------------------------------------------------------------------
    atapi_packet[ 0] = ATAPI_CMD_READ;
    atapi_packet[ 1] = 0x0;
    atapi_packet[ 2] = (lba >> 24) & 0xFF;
    atapi_packet[ 3] = (lba >> 16) & 0xFF;
    atapi_packet[ 4] = (lba >> 8) & 0xFF;
    atapi_packet[ 5] = (lba >> 0) & 0xFF;
    atapi_packet[ 6] = 0x0;
    atapi_packet[ 7] = 0x0;
    atapi_packet[ 8] = 0x0;
    atapi_packet[ 9] = numsects;
    atapi_packet[10] = 0x0;
    atapi_packet[11] = 0x0;
```

Now we should select the drive:

```
    // (II): Select the drive:
    // ------------------------------------------------------------------
    ide_write(channel, ATA_REG_HDDEVSEL, slavebit << 4);
```

400 nanoseconds delay after this select is a good idea:

```
    // (III): Delay 400 nanoseconds for select to complete:
    // ------------------------------------------------------------------
    for(int i = 0; i < 4; i++)
        ide_read(channel, ATA_REG_ALTSTATUS); // Reading the Alternate Sta
```

```
    // (IV): Inform the Controller that we use PIO mode:
    // ------------------------------------------------------------------
```

```
    ide_write(channel, ATA_REG_FEATURES, 0);         // PIO mode.
```

Tell the controller the size of the buffer

```
    // (V): Tell the Controller the size of buffer:
    // ---------------------------------------------------------------
    ide_write(channel, ATA_REG_LBA1, (words * 2) & 0xFF);   // Lower Byte
    ide_write(channel, ATA_REG_LBA2, (words * 2) >> 8);   // Upper Byte of
```

Now that we want to send the packet, we should first send the command "Packet":

```
    // (VI): Send the Packet Command:
    // ---------------------------------------------------------------
    ide_write(channel, ATA_REG_COMMAND, ATA_CMD_PACKET);      // Send the

    // (VII): Waiting for the driver to finish or return an error code:
    // ---------------------------------------------------------------
    if (err = ide_polling(channel, 1)) return err;         // Polling and

    // (VIII): Sending the packet data:
    // ---------------------------------------------------------------
    asm("rep   outsw" : : "c"(6), "d"(bus), "S"(atapi_packet));   // Send
```

Here we cannot poll. We should wait for an IRQ, then read the sectors. These two operations should be repeated for each sector.

```
    // (IX): Receiving Data:
    // ---------------------------------------------------------------
    for (i = 0; i < numsects; i++) {
       ide_wait_irq();                  // Wait for an IRQ.
       if (err = ide_polling(channel, 1))
          return err;      // Polling and return if error.
       asm("pushw %es");
       asm("mov %%ax, %%es"::"a"(selector));
       asm("rep insw"::"c"(words), "d"(bus), "D"(edi));// Receive Data.
       asm("popw %es");
       edi += (words * 2);
    }
```

Now we should wait for an IRQ and poll until the Busy and DRQ bits are clear:

```
    // (X): Waiting for an IRQ:
    // ---------------------------------------------------------------
```

```
   ide_wait_irq();

   // (XI): Waiting for BSY & DRQ to clear:
   // -------------------------------------------------------------------
   while (ide_read(channel, ATA_REG_STATUS) & (ATA_SR_BSY | ATA_SR_DRQ))
      ;

   return 0; // Easy, ... Isn't it?
}
```

Reading from an ATA/ATAPI Drive

```
void ide_read_sectors(unsigned char drive, unsigned char numsects, unsigr
                      unsigned short es, unsigned int edi) {

   // 1: Check if the drive presents:
   // ==================================
   if (drive > 3 || ide_devices[drive].Reserved == 0) package[0] = 0x1;

   // 2: Check if inputs are valid:
   // ==================================
   else if (((lba + numsects) > ide_devices[drive].Size) && (ide_devices[
      package[0] = 0x2;                     // Seeking to invalid positic

   // 3: Read in PIO Mode through Polling & IRQs:
   // ============================================
   else {
      unsigned char err;
      if (ide_devices[drive].Type == IDE_ATA)
         err = ide_ata_access(ATA_READ, drive, lba, numsects, es, edi);
      else if (ide_devices[drive].Type == IDE_ATAPI)
         for (i = 0; i < numsects; i++)
            err = ide_atapi_read(drive, lba + i, 1, es, edi + (i*2048));
      package[0] = ide_print_error(drive, err);
   }
}
// package[0] is an entry of an array. It contains the Error Code.
```

Writing to an ATA drive

```
void ide_write_sectors(unsigned char drive, unsigned char numsects, unsig
                       unsigned short es, unsigned int edi) {

   // 1: Check if the drive presents:
   // ==================================
   if (drive > 3 || ide_devices[drive].Reserved == 0)
```

```
      package[0] = 0x1;      // Drive Not Found!
   // 2: Check if inputs are valid:
   // ==================================
   else if (((lba + numsects) > ide_devices[drive].Size) && (ide_devices[
      package[0] = 0x2;                     // Seeking to invalid positic
   // 3: Read in PIO Mode through Polling & IRQs:
   // ============================================
   else {
      unsigned char err;
      if (ide_devices[drive].Type == IDE_ATA)
         err = ide_ata_access(ATA_WRITE, drive, lba, numsects, es, edi);
      else if (ide_devices[drive].Type == IDE_ATAPI)
         err = 4; // Write-Protected.
      package[0] = ide_print_error(drive, err);
   }
}
```

Ejecting an ATAPI Drive

```
void ide_atapi_eject(unsigned char drive) {
   unsigned int   channel      = ide_devices[drive].Channel;
   unsigned int   slavebit      = ide_devices[drive].Drive;
   unsigned int   bus      = channels[channel].Base;
   unsigned int   words      = 2048 / 2;               // Sector Size in
   unsigned char  err = 0;
   ide_irq_invoked = 0;

   // 1: Check if the drive presents:
   // ==================================
   if (drive > 3 || ide_devices[drive].Reserved == 0)
      package[0] = 0x1;      // Drive Not Found!
   // 2: Check if drive isn't ATAPI:
   // ==================================
   else if (ide_devices[drive].Type == IDE_ATA)
      package[0] = 20;         // Command Aborted.
   // 3: Eject ATAPI Driver:
   // ============================================
   else {
      // Enable IRQs:
      ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN = ide_ir

      // (I): Setup SCSI Packet:
      // ------------------------------------------------------------------
      atapi_packet[ 0] = ATAPI_CMD_EJECT;
      atapi_packet[ 1] = 0x00;
      atapi_packet[ 2] = 0x00;
      atapi_packet[ 3] = 0x00;
      atapi_packet[ 4] = 0x02;
```

```
      atapi_packet[ 5] = 0x00;
      atapi_packet[ 6] = 0x00;
      atapi_packet[ 7] = 0x00;
      atapi_packet[ 8] = 0x00;
      atapi_packet[ 9] = 0x00;
      atapi_packet[10] = 0x00;
      atapi_packet[11] = 0x00;

      // (II): Select the Drive:
      // ------------------------------------------------------------------
      ide_write(channel, ATA_REG_HDDEVSEL, slavebit << 4);

      // (III): Delay 400 nanosecond for select to complete:
      // ------------------------------------------------------------------
      for(int i = 0; i < 4; i++)
         ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Statu

      // (IV): Send the Packet Command:
      // ------------------------------------------------------------------
      ide_write(channel, ATA_REG_COMMAND, ATA_CMD_PACKET);      // Send t

      // (V): Waiting for the driver to finish or invoke an error:
      // ------------------------------------------------------------------
      err = ide_polling(channel, 1);            // Polling and stop if er

      // (VI): Sending the packet data:
      // ------------------------------------------------------------------
      else {
         asm("rep   outsw"::"c"(6), "d"(bus), "S"(atapi_packet));// Send
         ide_wait_irq();                  // Wait for an IRQ.
         err = ide_polling(channel, 1);            // Polling and get err
         if (err == 3) err = 0; // DRQ is not needed here.
      }
      package[0] = ide_print_error(drive, err); // Return;
   }
}
```

When this method is invoked, the optical device on the given channel is ejected.

See Also

Wiki Pages

- Master Boot Record (x86)
- Partition Table (x86)

Threads

- How to w/r harddisk in pmode? (ASM Code from Dex)

(http://www.osdev.org/phpBB2/viewtopic.php?t=12268)
- ATA PIO code library (ASM code from XCHG) (http://www.osdev.org/phpBB2/viewtopic.php?t=15314)
- IDE Tutorial (C code from *mostafazizo*) (http://forum.osdev.org/viewtopic.php?f=1&p=167798#p167798)

External Links

- T13 (http://www.t13.org) -- The group that creates the ATA standard
- ATA-ATAPI (http://www.ata-atapi.com) -- Public Domain C driver sources (including SATA, Busmatering DMA, ATAPI), fairly good.
- HDD Guru (http://hddguru.com/content/en/documentation/) -- The actual ATA specs from the first one that was released in 1994 to the 7th one in 2003.
- Partitioning Primer (http://www.ranish.com/part/primer.htm) -- A .HTM file containing some information about partitioning.